# File System

**Team name:** Team 05

| Team member | Student ID | GitHub name |
|---|---|---|
| Babak Milani | 920122577 | babakmilani |
| Mozhgan Ahsant | 921771510 | AhsantMozhgan |
| Gurpreet Natt | 922883894 | gpreet2 |
| Bisum Tiwana | 920388011 | SpindlyGold019 |

1. **Filesystem Github Link:**
   https://github.com/CSC415-2023-Fall/csc415-filesystem-gpreet2

## 2. Filesystem description:

Our file management system comprises a variety of functions that streamline the initiation, establishment, and application of a file system. Developed in the C programming language, our team authored code addressing fundamental file system components, including the volume control block, bitmap free space structure, directory entries, and functions supporting command execution. Our bespoke file system is designed to encompass key functionalities found in the Linux operating system's file system.

Our file system consists of the foundational components found in Linux file systems, drawing inspiration from their intricate design principles. At the heart of our file system lies the Volume Control Block (VCB), a pivotal element ensuring system integrity. We implemented a process where the system examines the VCB signature, determining whether volume formatting is necessary.

Our system leverages an efficient bitmap-based free space management system, ensuring optimal block allocation. This mechanism employs binary representation to denote used and free blocks, ensuring optimal storage utilization.

During system initialization in the “fsInit.c” file, we established the root directory, enabling hierarchical data organization within the file system. This directory provides a fundamental structure for storing and navigating stored data, laying the groundwork for file and directory operations.

The user interface named “fsshell.c” file, offers a range of commands for directory and file operations. Commands like”'ls, md, rm, mv, and cd enable users to interact with the file system.

Additionally, in the “b_io.c” file, we implemented essential file Input/Output operations, facilitating file interactions within the system. This component offers users a reliable interface for reading from and writing to files within the system, enhancing the overall usability and interaction with stored data.

Our file system, managed through the driver program fsshell.c, encompasses a comprehensive range of functionalities akin to those found in mainstream Linux file systems. The driver program serves as the gateway, orchestrating interactions with users and seamlessly executing file and directory operations based on their inputs. At its core, the fsshell.c file encapsulates a meticulously crafted main function that initiates the file system's partition via the startPartitionSystem() function, extracting essential parameters like file name, volume size, and block size from the command line arguments. This meticulous initialization ensures the system is ready for seamless user interactions.

One of the distinguishing features lies in the versatility and depth of the implemented commands. The program offers a vast array of functionalities comparable to those of Linux systems. From simple directory listing ('ls') to complex file manipulation ('cp', 'mv'), each command is meticulously designed to emulate familiar Linux functionalities while catering to unique aspects of our file system's architecture.

The interactions are user-friendly and intuitive, aligning closely with the standard Linux terminal commands. What sets our driver program apart is its optimization and efficiency in executing these commands. For instance, when copying files between systems ('cp2fs' and 'cp2l'), our system optimizes the data transfer process, ensuring rapid and error-free file movement. Moreover, the implementation of file system statistics ('fs_stat') provides users with insightful data about the system's overall health and performance.

Beyond functionality, our driver program emphasizes robustness and system integrity. During initialization, it thoroughly examines the volume, ensuring that formatting is appropriately performed only when necessary. This attention to detail maintains the file system's stability. Moreover, the program adopts efficient memory management strategies, optimizing file reads and writes, and intelligently handling free space allocation. In summary, our driver program, fsshell.c, encapsulates a rich array of functionalities mirroring the Linux file system, yet finely tuned and optimized for robustness, efficiency, and user-friendliness.

**These are the core functions for our file system:**

**A. These initialization and exit functions are integral to establishing and maintaining the stability, functionality, and proper operation of the file system. They set the stage for core operations and ensure that the file system is properly initialized, functional during runtime, and gracefully exits without any data loss or inconsistencies.**

a. initGlobalVar(): This function initializes global variables used throughout the file system. It allocates memory and sets default values for variables essential for various operations and tracking within the file system. It might involve allocating memory for paths, directories, or critical data structures used across the file system. Setting initial pointers, flags, or status indicators could also be part of this initialization process.

b. freeGlobalVar(): This function is responsible for releasing memory occupied by global variables when they are no longer needed. It ensures efficient memory management by deallocating resources acquired during initialization. It may involve freeing dynamically allocated memory, closing open handles or file descriptors, and performing other cleanup operations to prevent memory leaks.

c. initFileSystem(): This function serves as the primary initialization routine for the entire file system. It sets up fundamental components such as the Volume Control Block (VCB), root directory, free space management, and other essential data structures. This function could include tasks like initializing the VCB with default values, creating the root directory entry, setting up the free space management system (like initializing a bitmap), and performing other crucial setup tasks required for the file system to function.

d. exitFileSystem(): This function performs cleanup operations and gracefully exits the file system. It ensures proper shutdown by freeing resources, closing open files or directories, and saving any necessary metadata or states before exiting. Tasks might involve freeing allocated memory, updating metadata or structures to reflect changes made during runtime, ensuring the integrity of data on disk, and closing any open connections or handles associated with the file system.

**B. These functions handle directory creation, manipulation, traversal, and maintenance within the file system, providing essential functionalities for managing directories, their contents, and working with the file system's structure:**

a. fs_mkdir(const char *pathname, mode_t mode): This function creates a new directory within the file system at the specified path. It is responsible for creating a directory entry within the file system's structure, initializing it with necessary attributes (like permissions, timestamps, etc.), and allocating space for the directory content.

b. fs_rmdir(const char *pathname): This function removes a directory from the file system. It deletes the specified directory entry and its associated content. It ensures that the directory is empty before removal and handles cleanup of allocated space or any related metadata.

c. fs_opendir(const char *pathname): This function opens a directory for reading. It sets up a directory handle, allowing subsequent reading or traversing of the contents within the specified directory.

d. fs_readdir(fdDir *dirp): This function retrieves information about the next directory entry within an open directory. It facilitates the iteration through directory entries, providing information about each file or subdirectory present within the directory handle.

e. fs_closedir(fdDir *dirp): This function closes an open directory handle. It is responsible for releasing resources associated with an open directory handle, finalizing directory reading operations, and freeing allocated memory.

f. fs_getcwd(char *buf, size_t size): This function retrieves the current working directory path. It returns the absolute path of the current working directory, storing it in the provided buffer.

g. fs_setcwd(char *buf): This function changes the current working directory. It sets the current working directory to the path specified by the provided buffer.

h. fs_isFile(char * path): This function checks if the given path refers to a file. It examines the specified path within the file system and determines if it corresponds to a file.

i. fs_isDir(char * path): This function checks if the given path refers to a directory. It determines whether the specified path in the file system points to a directory.

j. fs_delete(char* filename): This function removes a file from the file system. It deletes the specified file entry and its associated data from the file system.

**C. These path manipulation functions facilitate the breakdown of complex path strings into manageable components, assisting in directory navigation, file creation, deletion, and other file system operations**:

a. parsePath(const char *pathname): This function breaks down the given path into individual components.It parses the provided path, separating it into distinct elements that represent different directories or file names along the path. It dissects the path to identify each segment and their relationships within the file system structure.

b. getParentDirectory(const char *pathname): This function extracts the parent directory from a given path. It isolates the parent directory of the specified path, allowing the identification of the directory where the path is located.

c. getLastPathElement(const char *pathname): This function retrieves the last element from a path. It extracts the final component or file name from a given path, enabling the identification of the actual file or directory being referenced at the end of the path.

**D. These file operation functions enable users to interact with files by performing read, write, seek, and close operations on them:**

a. b_open (char * filename, int flags): This function opens a file with specified attributes. This function initiates the process of opening a file within the file system. It accepts parameters such as the file name and flags that denote various file attributes (e.g., read-only, write-only, create if not exists, etc.). It's responsible for initializing the file for further operations.

b. b_read (b_io_fd fd, char * buffer, int count): This function reads data from an open file. This function facilitates the reading of data from an already opened file. It takes in parameters such as the file descriptor, a buffer to store the read data, and the count of bytes to read. It reads the specified number of bytes from the file and stores them in the provided buffer.

c. b_write (b_io_fd fd, char * buffer, int count): This function writes data to an open file. This function enables the writing of data into an already opened file. It receives the file descriptor, the buffer containing the data to be written, and the count of bytes to write. It writes the specified number of bytes from the buffer into the file.

d. b_seek (b_io_fd fd, off_t offset, int whence): This function moves the file pointer to a specified location.This function facilitates the repositioning of the file pointer within an open file. It takes in parameters including the file descriptor, the offset specifying the new position, and a 'whence' parameter determining how the offset should be interpreted (e.g., beginning, current position, end). It allows seeking to different positions within a file.

e. b_close (b_io_fd fd): This function closes an open file. This function is responsible for closing an open file within the file system. It takes the file descriptor as input and completes the file operation, ensuring that the file is no longer open and resources associated with it are properly released.

**E. These functions collectively provide essential functionalities related to file and directory management within the file system. They allow users to navigate directories, check the nature of paths (file or directory), set the current working directory, and delete files as needed.**

a. fs_getcwd(char *buf, size_t size): This function retrieves the current working directory. This function retrieves the absolute pathname of the current working directory and stores it in the provided buffer. The "buf" parameter is a pointer to a character array where the path will be stored, and 'size' denotes the size of the buffer.

b. fs_setcwd(char *buf): This function sets the current working directory. This function changes the current working directory to the path specified in the 'buf' parameter. It allows users to navigate through the directory structure by changing the context of the current working directory.

c. fs_isFile(char *path): This function checks if a given path points to a file. This function verifies whether the specified 'path' refers to a file within the file system. It returns a boolean value (1 for true, 0 for false) indicating whether the path leads to a file.

d. fs_isDir(char *path): This function checks if a given path points to a directory. It is similar to fs_isFile, this function determines if the provided path points to a directory within the file system. It returns a boolean value (1 for true, 0 for false) indicating whether the path leads to a directory.

e. fs_delete(char* filename): This function deletes a specified file. This function removes the file specified by 'filename' from the file system. It's responsible for deallocating the space occupied by the file and updating the file system metadata to reflect the removal of the file.

**F. This function plays a crucial role in enabling users or applications to obtain detailed insights into the properties and attributes of files or directories within the file system. It aids in monitoring and understanding the characteristics and usage patterns of various elements stored in the file system:**

a. fs_stat(const char *path, struct fs_stat *buf): The function provides a convenient way for users or applications to gather essential information about a file or directory, such as

its size, timestamps for access, modification, and other relevant metadata associated with the specified path within the file system. Typically, upon successful execution, this function populates the provided `struct fs_stat *buf` with the gathered statistics, allowing users or programs to access and utilize this information as required. This function retrieves file system statistics. This function gathers various statistical information related to the file or directory specified by the 'path' parameter and stores this information in the 'buf' structure. This structure typically contains fields such as:
- st_size: Total size of the file in bytes.
- st_blksize: Blocksize for file system I/O.
- st_blocks: Number of 512-byte blocks allocated.
- st_accesstime: Time of the last access to the file.
- st_modtime: Time of the last modification of the file.
- st_createtime: Time of the last status change of the file.

**These are the core structures for our file system:**

**G. These core structures collectively form the foundation of our file system, providing the necessary framework for organizing, managing, and accessing data stored within the system:**

a. Volume Control Block (VCB): The VCB acts as a metadata structure responsible for managing and organizing the entire volume.
   - Attributes: Includes fields such as:
     - signature: Unique identifier marking the VCB.
     - numBlocks: Total number of blocks in the volume.
     - blockSize: Size of each block in bytes.
     - freeSpace: Number of free blocks available.
     - freeSpaceBitMap: Memory pointer for the free space bitmap.
     - bitMapByteSize: Size of the bitmap in bytes.
     - RootDir: Index of the root directory block.

b. Directory Entry Structure: Represents the structure used for storing information about individual directories and files within the file system.
   - Attributes: Typically includes fields like:
   - d_reclen: Length of the directory entry.
   - fileType: Type of file (regular file, directory).
   - d_name: Name of the file or directory.

c. File System Data Structures:
   - Bitmap-Based Free Space Management: Efficiently tracks the allocation status of data blocks within the volume. Utilizes a bitmap structure where '1' indicates a used block, and '0' signifies a free block, allowing for efficient allocation and management of available space.

d. File Descriptors and Pointers: Used for file manipulation and tracking the position within files or directories.

- Attributes: Contains elements like:
  - dirEntryPosition: Position within the directory entry.
  - directoryStartLocation: Starting LBA (Logical Block Address) of a directory.
  - dirPointer: Pointer to directory entry information.
  - dirSize: Size of the directory.
  - fileIndex: Index pointing to specific files.

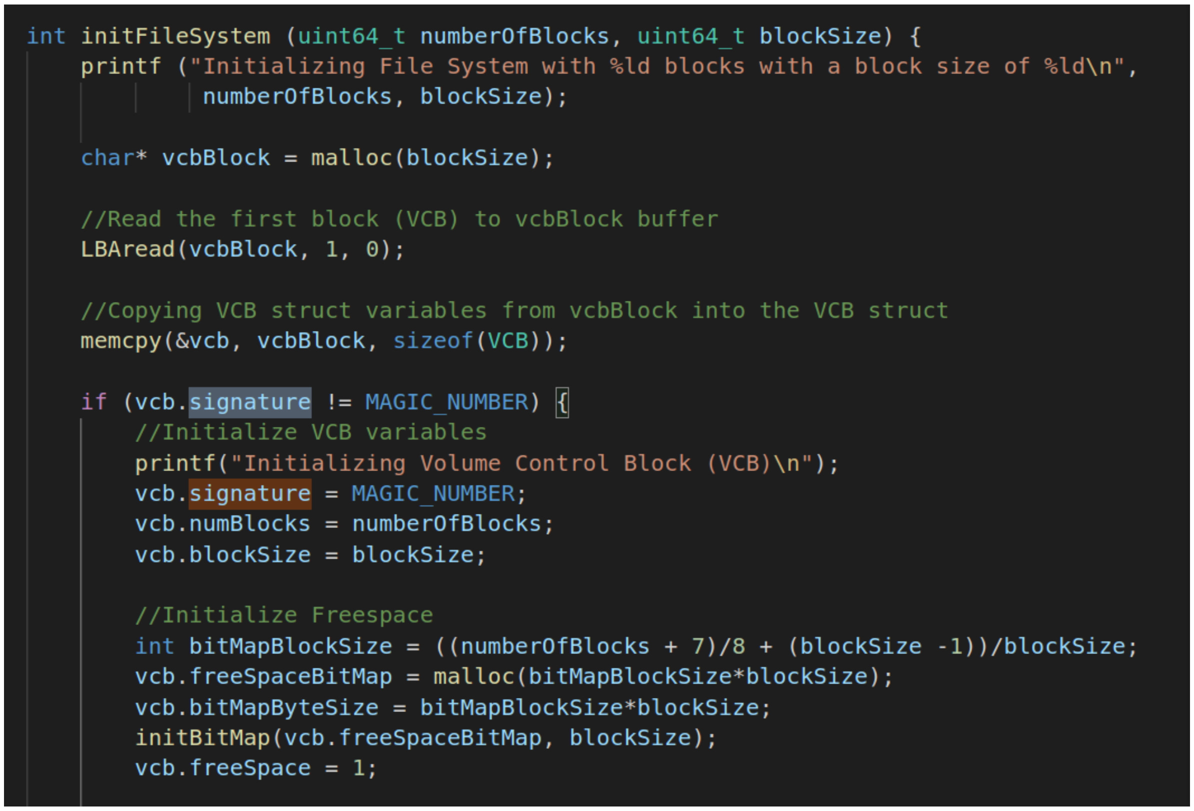
e. Path Parsing and Manipulation Structures: Facilitates parsing and manipulation of file paths.
   - Elements: Includes structures like pathInfo containing:
   - DEPointer: Pointer to directory entry information.
   - value: A value indicating successful parsing.
   - path: Path string or elements.

f. File System Statistics Structure: Stores statistics related to files and directories.
   - Attributes: Contains fields like st_size, st_blksize, st_blocks, st_accesstime, st_modtime, st_createtime.

3. **Issues:**

One of the issues that we had was Determining VCB Validity: How do we ascertain the validity of the Volume Control Block and decide whether initialization is necessary?

Solution:
Our system relies on a hardcoded signature within the Volume Control Block. Initialization Check: During file system initialization, we retrieve the VCB from the disk and compare its signature with our hardcoded value. If the signatures match, indicating that the VCB is already formatted and valid, we proceed without the need for further formatting. If the signatures don't match, signaling that the VCB needs formatting, we trigger the initialization process. This involves formatting the VCB and initializing other necessary components for the file system to operate smoothly.



```
int initFileSystem (uint64_t numberOfBlocks, uint64_t blockSize) {
    printf ("Initializing File System with %ld blocks with a block size of %ld\n",
                numberOfBlocks, blockSize);

    char* vcbBlock = malloc(blockSize);

    //Read the first block (VCB) to vcbBlock buffer
    LBAread(vcbBlock, 1, 0);

    //Copying VCB struct variables from vcbBlock into the VCB struct
    memcpy(&vcb, vcbBlock, sizeof(VCB));

    if (vcb.signature != MAGIC_NUMBER) {
        //Initialize VCB variables
        printf("Initializing Volume Control Block (VCB)\n");
        vcb.signature = MAGIC_NUMBER;
        vcb.numBlocks = numberOfBlocks;
        vcb.blockSize = blockSize;

        //Initialize Freespace
        int bitMapBlockSize = ((numberOfBlocks + 7)/8 + (blockSize -1))/blockSize;
        vcb.freeSpaceBitMap = malloc(bitMapBlockSize*blockSize);
        vcb.bitMapByteSize = bitMapBlockSize*blockSize;
        initBitMap(vcb.freeSpaceBitMap, blockSize);
        vcb.freeSpace = 1;
```

The second of the issues that we had was with bit manipulation errors. We solved it by creating unit tests for ‘setABit’ and ‘clearABit’ in mapping.c. The test cases included:

- Setting and clearing the first and last bits in a byte.
- Setting and clearing every bit position within a byte.
- Edge cases, such as setting a bit that is already set or clearing a bit that is already clear.
- Sequential sets and clears to simulate typical usage patterns.

## 4. How things work

**An introduction to the commands available within our driver program:**

These commands offer users a range of functionalities, enabling them to interact with the file system efficiently, manage directories, and perform file operations seamlessly.

1. **ls** (List): Lists the contents (files and directories) within the current directory. The `ls` command lists the contents of a directory within the file system. It displays filenames, their associated attributes, and whether they are directories or regular files. Users can traverse directories and retrieve detailed information about their contents.

   - Usage: ls [directory_name]

   - Example: ls /documents

2. **cp** (Copy): Copies a file or directory from one location to another. The `cp` command copies files from a source to a destination within the file system. It facilitates file duplication while preserving attributes and content integrity.

   - Usage:  cp [source_path] [destination_path]

   - Example: cp /documents/file.txt /backup

3. **mv** (Move): Moves a file or directory to a different location. The `mv` command moves files from one location to another within the file system. It is used for renaming files or transferring them between directories.

   - Usage: mv [source_path] [destination_path]

   - Example: mv /documents/file.txt /archive

4. **md** (Make Directory): Creates a new directory. The `md` command creates a new directory within the file system. It enables users to organize data by establishing a hierarchical structure.

   - Usage: md [directory_name]

   - Example: md new_folder

5. **rm** (Remove): Deletes a file or directory. The `rm` command deletes files or directories from the file system, freeing up space and allowing users to manage their storage efficiently.

- Usage: rm [file_name or directory_name]

- Example: rm old_file.txt

6. **touch**: Creates a new empty file. The `touch` command creates new files within the file system. It allows users to quickly generate empty files or update file timestamps.

- Usage: touch [file_name]

- Example: touch new_file.txt

7. **cat** (Concatenate): Displays the contents of a file. The `cat` command displays the contents of a file within the file system. While its functionality might be limited, it provides users with a quick view of file content.

- Usage: cat [file_name]

- Example: cat data.txt

8. **cd** (Change Directory): Changes the current working directory. The `cd` command changes the current working directory within the file system, allowing users to navigate through directories.

- Usage: cd [directory_name]

- Example: cd documents

9. **pwd** (Print Working Directory): Prints the current working directory. The `pwd` command displays the current working directory path within the file system, providing users with their location within the directory structure.

- Usage: pwd

- Example: pwd

10. **history:** Displays command history. The `history` command displays a history of previous commands executed within the file system, facilitating users in reviewing their actions.

- Usage: history

- Example: history

11. **help**: Provides assistance and usage instructions. The `help` command provides users with information about available commands, syntax, and their functionalities within the file system.

- Usage: help [command_name]

- Example: help ls

12. **cp2l** (Copy to Linux): Copies a file from the file system to the Linux file system. The `cp2l` command copies files from the file system to the Linux file system, enabling data transfer between the two environments seamlessly.

- Usage: cp2l [file_name]

- Example: cp2l file_from_fs.txt

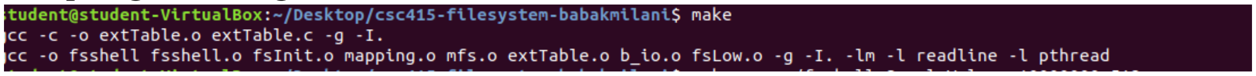
13. **cp2fs** (Copy to File System): Copies a file from the Linux file system to the file system. The `cp2fs` command performs the inverse operation of `cp2l`, copying files from the Linux file system to the file system created by your project. It supports bidirectional file transfer.

- Usage: cp2fs [file_name]

- Example: cp2fs file_from_linux.txt

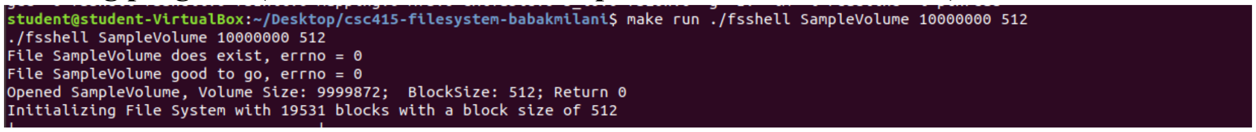
## 5. Screenshots

## Compiling/Executing "make"



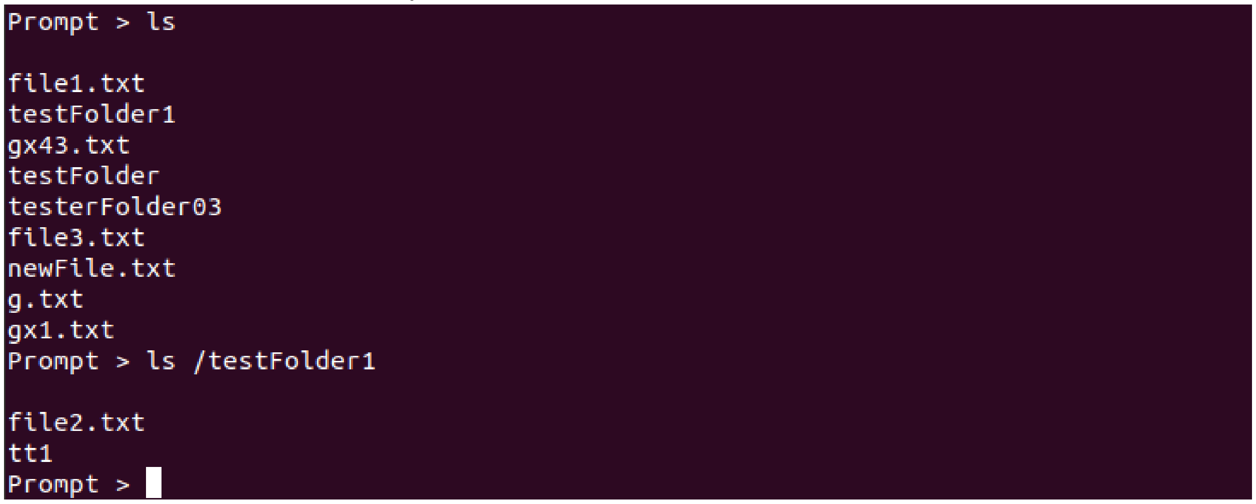
```
tudent@student-VirtualBox:~/Desktop/csc415-filesystem-babakmilani$ make
cc -c -o extTable.o extTable.c -g -I.
cc -o fsshell fsshell.o fsInit.o mapping.o mfs.o extTable.o b_io.o fsLow.o -g -I. -lm -l readline -l pthread
```

## Running program ("make run ./fsshell SampleVolume 10000000 512")



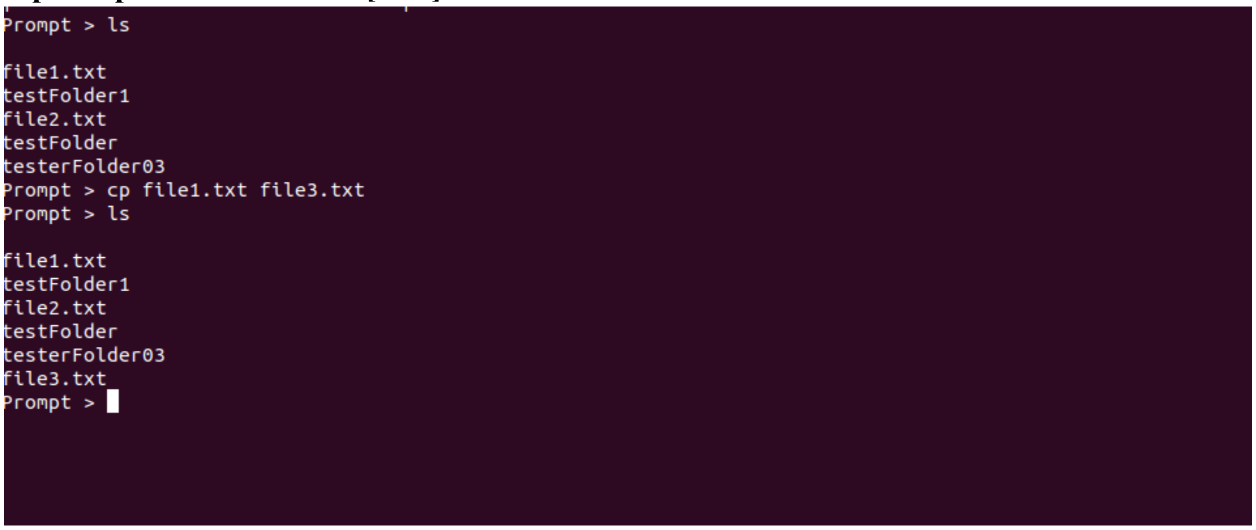
```
student@student-VirtualBox:~/Desktop/csc415-filesystem-babakmilani$ make run ./fsshell SampleVolume 10000000 512
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
```

## "ls" Lists the file in a directory



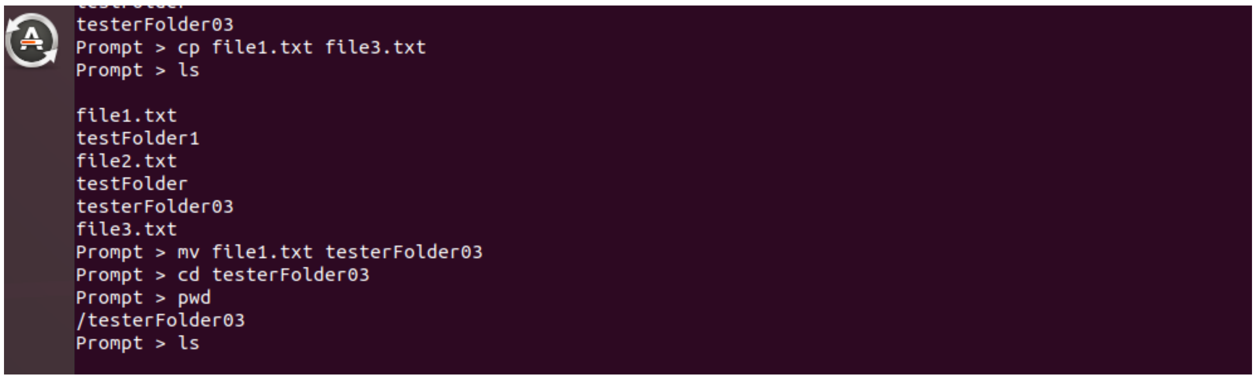
```
Prompt > ls

file1.txt
testFolder1
gx43.txt
testFolder
testerFolder03
file3.txt
newFile.txt
g.txt
gx1.txt
Prompt > ls /testFolder1

file2.txt
tt1
Prompt > 
```

## "cp" Copies a file - source [dest]



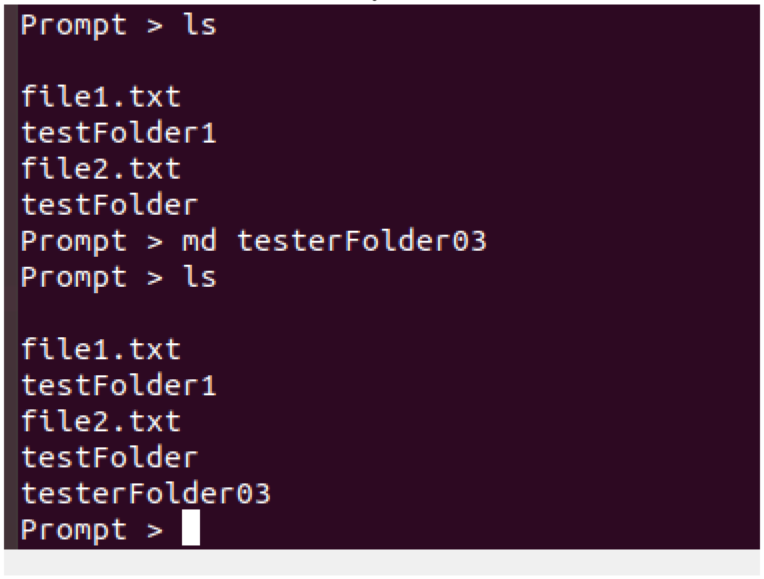
```
Prompt > ls

file1.txt
testFolder1
file2.txt
testFolder
testerFolder03
Prompt > cp file1.txt file3.txt
Prompt > ls

file1.txt
testFolder1
file2.txt
testFolder
testerFolder03
file3.txt
Prompt >
```

## "mv" Moves a file - source dest



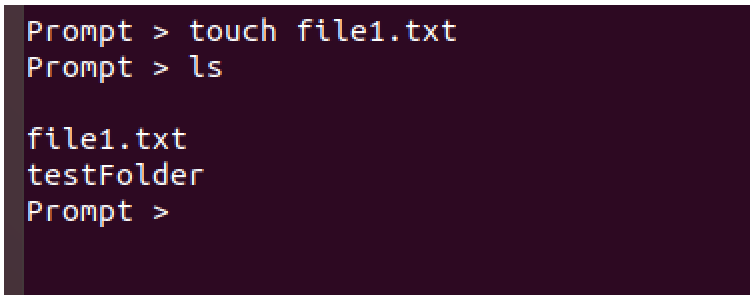
```
testerFolder03
Prompt > cp file1.txt file3.txt
Prompt > ls

file1.txt
testFolder1
file2.txt
testFolder
testerFolder03
file3.txt
Prompt > mv file1.txt testerFolder03
Prompt > cd testerFolder03
Prompt > pwd
/testerFolder03
Prompt > ls
```

**"md" Make a new directory**



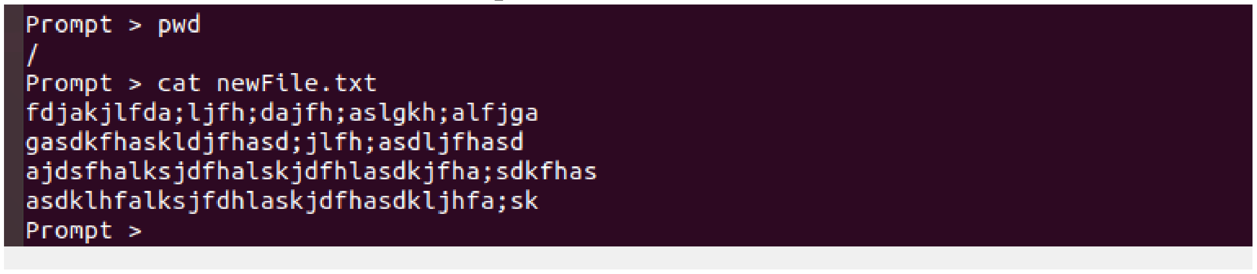
```
Prompt > ls

file1.txt
testFolder1
file2.txt
testFolder
Prompt > md testerFolder03
Prompt > ls

file1.txt
testFolder1
file2.txt
testFolder
testerFolder03
Prompt >
```

**"touch" Touches/Creates a file**



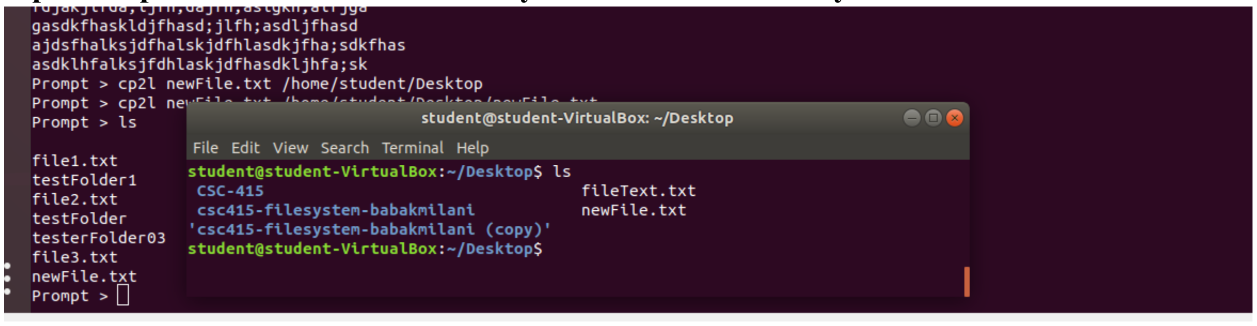
```
Prompt > touch file1.txt
Prompt > ls

file1.txt
testFolder
Prompt >
```

**"cat" Limited version of cat that displace the file to the console**



```
Prompt > pwd
/
Prompt > cat newFile.txt
fdjakjlfda;ljfh;dajfh;aslgkh;alfjga
gasdkfhaskldjfhasd;jlfh;asdljfhasd
ajdsfhalksjdfhalskjdfhlasdkjfha;sdkfhas
asdklhfalksjfdhlaskjdfhasdkljhfa;sk
Prompt >
```

**"cp2l" Copies a file from the test file system to the linux file system"**

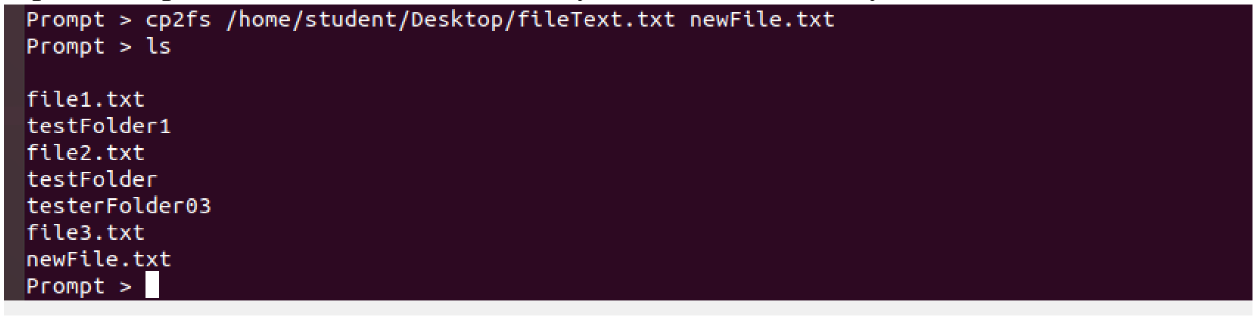


```
[illegible]
gasdkfhaskldjfhasd;jlfh;asdljfhasd
ajdsfhalksjdfhalskjdfhlasdkjfha;sdkfhas
asdklhfalksjfdhlaskjdfhasdkljhfa;sk
Prompt > cp2l newFile.txt /home/student/Desktop
Prompt > cp2l newFile.txt /home/student/Desktop/newFile.txt
Prompt > ls

file1.txt
testFolder1
file2.txt
testFolder
testerFolder03
file3.txt
newFile.txt
Prompt >
```

student@student-VirtualBox: ~/Desktop
File Edit View Search Terminal Help

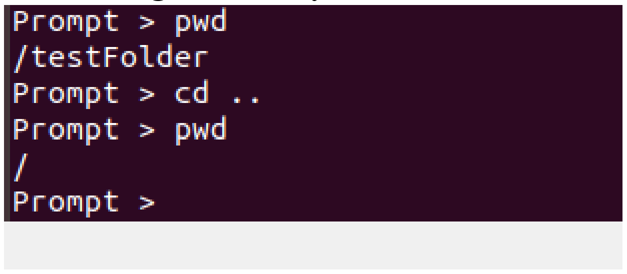
```
student@student-VirtualBox:~/Desktop$ ls
 CSC-415                                      fileText.txt
 csc415-filesystem-babakmilani                newFile.txt
'csc415-filesystem-babakmilani (copy)'
student@student-VirtualBox:~/Desktop$
```

**"cp2fs" Copies a file from the Linux file system to the test file system**



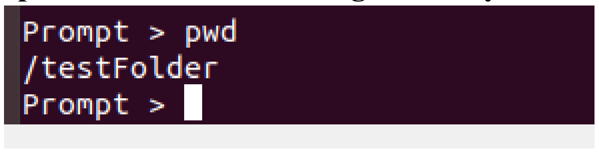
```
Prompt > cp2fs /home/student/Desktop/fileText.txt newFile.txt
Prompt > ls

file1.txt
testFolder1
file2.txt
testFolder
testerFolder03
file3.txt
newFile.txt
Prompt >
```

**"cd" Changes directory**



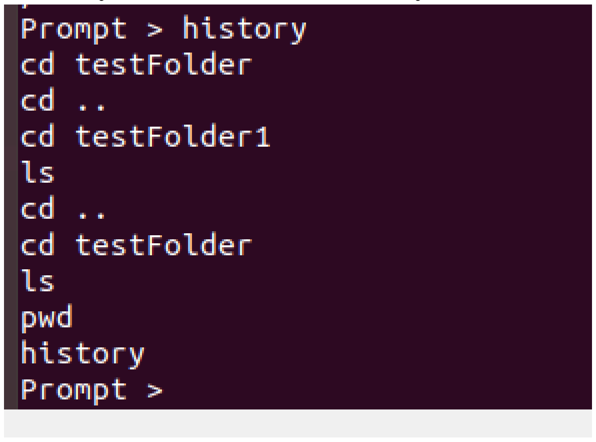
```
Prompt > pwd
/testFolder
Prompt > cd ..
Prompt > pwd
/
Prompt >
```

**"pwd" Prints the working directory**



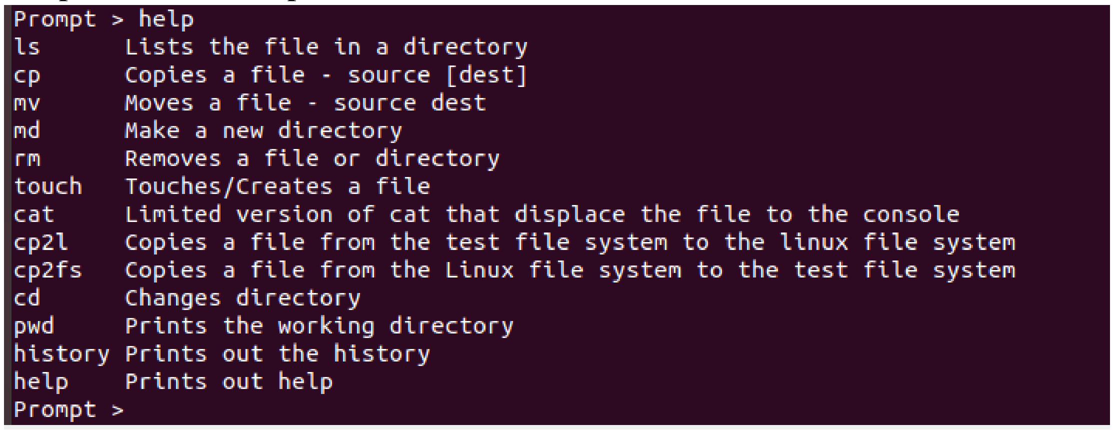
```
Prompt > pwd
/testFolder
Prompt >
```

**"history" Prints out the history**



```
Prompt > history
cd testFolder
cd ..
cd testFolder1
ls
cd ..
cd testFolder
ls
pwd
history
Prompt >
```

**"help" Prints out help**



```
Prompt > help
ls      Lists the file in a directory
cp      Copies a file - source [dest]
mv      Moves a file - source dest
md      Make a new directory
rm      Removes a file or directory
touch   Touches/Creates a file
cat     Limited version of cat that displace the file to the console
cp2l    Copies a file from the test file system to the linux file system
cp2fs   Copies a file from the Linux file system to the test file system
cd      Changes directory
pwd     Prints the working directory
history Prints out the history
help    Prints out help
Prompt >
```